International Journal of Exploring Emerging Trends in Engineering (IJEETE)
Vol. 03, Issue 02, Mar-Apr, 2016 Pg. 122 - 125 WWW.IJEETE.COM

AUTOMATIC PARALLELIZATION OF C PROGRAMS FOR CLUSTER AND GPU COMPUTING

[1]Ayesha Parveen Khanbu, [2]Nida Patankar, [3]Harshala Bhoir, [4]Amroz Siddiquiy
Department of Computer Engineering
Fr. Conceicao Rodrigues Institute of Technology, Vashi
Navi Mumbai, India

ABSTRACT - *A C program essentially runs sequentially and does not inherently support parallel computation. Keeping the throughput in mind, we present two ways of automatic paral-lelization of C programs - 1) by executing the program on a commodity cluster where our tool transforms a program into its parallelized equivalent which uses OpenMPI libraries as a com-munication mechanism between processors, 2) by transforming the sequential C program into an equivalent CUDA C program to significantly enhance the speed of execution of the program by leveraging data parallelism of GPUs. Currently we have limited our scope to integer arrays and for loops.*

Keywords— Automatic Parallelization, Cluster Computing,CUDA, GPU Computing, OpenMPI, Polyhedral Model.

INTRODUCTION

Parallel processing has been a topic of study and research for a number of years. However, loop level parallelism has gained importance only in the past few years. Due to the ever growing trend of multi-core architecture, parallel programming is important as well as interesting. But, it is seen that parallel programming is a difficult chore requiring great efforts from the programmer's side. One conclusive elucidation is that there is a need for automatic parallelization.

Automatic parallelization is a mechanism of automatically converting a sequential program to a version that can directly run on multiple processing elements without changing the meaning of the program. Automatic parallelization is typically performed by a compiler, at a high level where most of the information needed is available. Computing power can be used effectively if the programmers write only the sequential codes and leave the task of parallelization to the compiler. An auto-parallelizer is a program that generates parallel programs that produce the same result as sequential programs. This paper deals with compile-time automatic parallelization and primarily targets shared memory parallel architectures for which auto-parallelization is significantly easier.

Currently well accepted methods of parallel programming, such as OpenMP or MPI, are essentially extensions to existing languages, like C or Fortran. On one hand, it allows reuse of an existing code base while on the other hand, it requires both the compiler and programmers to deal with languages that were not originally designed for parallelism. The major challenges involved in design and implementation of such a tool include side-effects of function calls, finding alias

Fig. 1: Master-Slave architecture

variables, dependency between statements, etc. Additionally the tool has to deal with the variety in coding styles, length and number of files. It is also important to take into consideration the amount of inherent parallelism the application provides.

A. Polyhedral model

The polyhedral model is a robust mathematical framework for automatic optimization and parallelization which is also known as the polytope model. Many scientific and engineering applications spend most of their execution time in nested loops. It is based on an algebraic representation of programs, allowing to construct and search for complex sequences of optimiza-tions. It treats each loop iteration within nested loops as lattice points inside mathematical objects or a well defined space called the polyhedron as seen in Figure 2. Also, the most im-portant concept in the polyhedron are the integer

points, since loop iterators are integers and travel by an integral amount along the axes in the space. Depending merely on concepts of linear algebra and integer linear programming, it is possible to reason about the correctness of complex loop transformations. It successfully performs affine transformations or more general non-affine transformations such as tiling on the polyhedron,

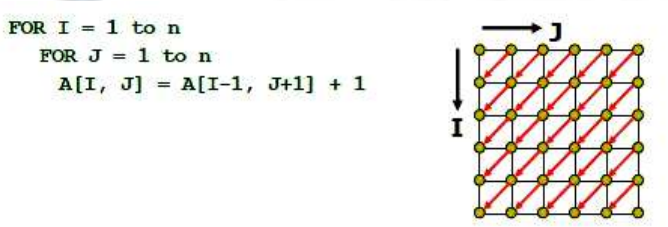

Fig. 2: Polyhedral model representation of sample code

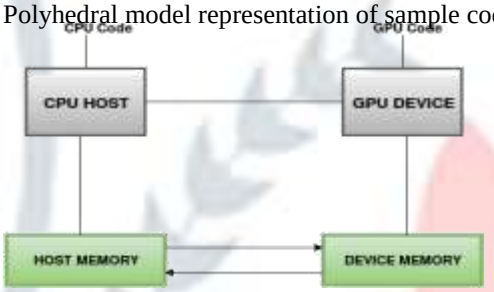

Fig. 3: CUDA architecture

and then converts the transformed polyhedron into equivalent, but optimized (depending on targeted optimization goal), loop nests through polyhedra scanning[10]. The core transformation framework mainly works by finding affine transformations for efficient tiling

The task of program optimization (often for parallelism and locality) in the polyhedral model may be viewed in terms of three phases:

1. static dependence analysis of the input program transformations in the polyhedral abstraction
2. generation of code for the transformed program

B. Polyhedral Dependencies

Two iterations are said to be dependent if they access the same memory location. Also, one of them should be a write operation[6]. There are different types of dependencies namely, Read-After-Write (RAW) dependencies, Write-After-Read (WAR) dependences that is,if a write operation is performed on a memory location followed by a read. Similarly, WAW and RAR are also two of them. Any reordering will only be legal if does not violate the dependencies, i.e. one is allowed to change the order in which operations are performed as long as the transformed program has the same execution order with respect to the dependent iterations.

C. *OpenMPI*

OpenMPI is a Message Passing Interface (MPI) library project. It is used by many TOP 500 supercomputers including the world's most efficient supercomputer till date. MPI is a language-independent communications protocol used for programming parallel computers. In a master-slave architecture, the master sends new task data to a slave whenever the prior task is completed using the MPI functions. It provides a complete set of functions used for communication between the master and slaves connected to each other via., a switch (in our case, a KVM switch) like MPI send, MPI recv etc. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

2*CUDA*

CUDA is a parallel computing platform developed by NVIDIA and introduced in 2007[12]. It enables dramatic in-crease in computing performance by making use of the power of the graphics processing unit (GPU). GPUs are enabled with significantly elevated processing power but most of the time these GPUs remain idle. Today, most of the machines come with NVIDIA graphics card which contains GPU having various processing cores. It is basically used during execution of gaming image processing and graphics type of applications. The computing capability of the GPU available can be properly utilized during execution of applications outside the graphics world. Programming GPU is a very complex and difficult task as compared to programming CPU and parallel programming models. A GPU has a multi-core architecture that is capable of running thousands of threads concurrently and hence the application performance can be enhanced when a GPU is deployed.

CUDA is a special programming language that is designed to program NVIDIA GPUs. But, programming with CUDA is an extremely difficult task for novice users. Even though the CUDA

programming model offers a user friendly interface, programming GPGPUs is tough and error prone as compared to programming parallel programming models like OpenMPI. Manually programming for data transfer to and from the device and host is a time consuming and difficult task. A CUDA architecture is as shown in Figure 3 where initially program is submitted by the user on the CPU host. The variables are allocated space in the GPU device, followed by execution on the device. then, copying the results back to the CPU host anf finally freeing the GPU memory.

II. PROPOSED SYSTEM

Figure 4 depicts the system diagram for our tool CtoCUD-ATranslator, containing input, output and basic system blocks. This implementation is based on the techniques proposed by Sven Verdoolaege[11]. CtoCUDATranslator converts a se-quential C programs provided as input into an equivalent CUDA program, which is generated as the output. In our

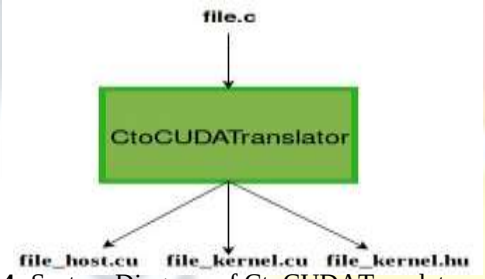

Fig. 4: System Diagram of CtoCUDATranslator

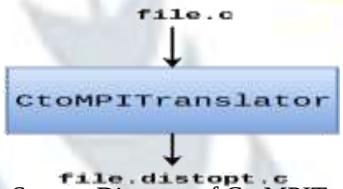

Fig. 5: System Diagram of CtoMPITranslator

current implementation, a programmer surrounds the code to be parallelized between #pragma scop and #pragma endscop. In 4, the input program named file.c is passed to our tool. The tool then configures the grid required to execute the program on GPU cores and generates three files as output - filename kernel.hu, filename kernel.cu and filename host.cu. The parallelized version of the code between #pragma scop and #pragma endscop from filename.c are stored as a function in filename kernel.cu and this is executed on the GPU. Calls to this function are made from filename host.cu, which is executed on the host (CPU). The program running on the host is responsible for allocating memory, invoking the function and copying memory between the host and GPU. As a result, when the GPU finishes execution, the results are passed to the host.

Figure 5 shows the system diagram for another tool called CtoMPITranslator, containing input, output and basic system block. This implementation of CtoMPITranslator is based on the techniques proposed by Uday Bondhugula [1][3] for converting sequential C program into an equivalent OpenMPI-based program. The input to this tool is a serial C program and output is an equivalent parallel OpenMPI-based program that can be executed in a distributed processing environ-ment as shown in Figure 1. As with CtoCUDATranslator, the programmer places the code to be parallelized between #pragma scop and #pragma endscop. CtoMPITranslator takes as input the sequential C code in filename.c and generates a file named filename.distopt.c as shown in 5. The code not withing #pragmas is just copied as is to the generated filename.distopt.c file. When compiled to a binary executable, filename.distopt.c can be run on a cluster of single or multicore processors.

TESTING AND RESULTS

We tested our CtoMPITranslator tool on a Beowulf cluster of 4 nodes with Pentium 4 processors running Ubuntu 14.04 LTS. The running times for two benchmarks involving matrices of size NxN are shown in Table I.

We tested out CtoCUDATranslator on a machine with an Intel Core i7 CPU, 8 GB of RAM and an NVIDIA GPU. Our tool successfully generated the equivalent CUDA C program

TABLE I: Running time (in milliseconds) of serial and parallel execution on the cluster

nchmark		rial	rallel
atmul.c	6	1.981	1.345
atmul.c	2	25.144	4.950
atmul.c	24	218.999	15.296
idel.c	0	9.606	7.814
idel.c	00	76.435	90.657
idel.c	00	812.177	555.879

files for all benchmarks, which were then compiled and run using the NVIDIA CUDA Toolkit.

IV.CONCLUSION AND FUTURE SCOPE

The implemented system is divided into two parts.

First half is a tool which converts a serial C program to its equivalent Parallel C program using OpenMPI libraries. The second half of the project is also a tool which generates an equivalent CUDA C program without manual participation of the user in the entire process. It is found that both the parallel programs so generated are correct as observed during testing the generated output files. Both the tools are user friendly and need not proficiently know OpenMPI or CUDA programming to use the tool for conversion of serial C programs to parallel CUDA C or OpenMPI programs.

The system successfully makes good utilisation of cheap commodity-grade computers and GPU. It utilizes the commod-ity hardware by reducing the communication volume across the computers connected in the cluster. The system also helps to utilize the graphics cards available in the computer system to accelerate the general purpose applications and thus en-hancing the system performance. This project can be extended to parallelize programs based on the entire C language, or languages such as Java, C++ or even domain specific languages like Matlab, Mathematica, etc., most of which demand high computation power.

ACKNOWLEDGEMENTS

We are grateful to the Department of Computer Engineering, Fr. C.R.I.T. for giving us this opportunity to work in the parallel computation domain and also for providing all the necessary hardware for the cluster. We would like to thank Lakshmi Gadhikar for providing us the hardware for testing our CtoCUDATranslator tool. We would also like to thank Rohit Jha for his help and constant support during the de-velopment and for reviewing this paper.

REFERENCES

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimizer. In ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'08) , Tucson, AZ, USA, June 2008.

[2] Louis-Noel Pouchet, Cedric Bastoul , Albert Cohen, John Cavazos. Iterative Optimization in the Polyhedral Model:Part II, Multidimensional Time.

[3] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanu-jam, Atanas Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model.

[4] PLUTO, URL:http://www.ece.lsu.edu/jxr/pluto/index.html, Retrieved on: 21 June, 2015

[5] Tomofumi Yuki.Beyond shared memory loop parallelism in the polyhe-dral model.

[6] Uday Kumar Reddy Bondhugula.Effective automatic parallelization and locality optimization using the polyhedral model.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayap-pan.PLuTo: A Practical and Fully Automatic Polyhedral Program Op-timization System.

[8] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, Sanjay Rajopadhye, AlphaZ: A System for Design Space Exploration in the Polyhedral Model*

[9] Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayap- pan, P.: Hybrid iterative and model-driven optimization in the polyhedral model.

[10] Polyhedral Compilation, URL:http://polyhedral.info/

AUTHOR's BIBLOGRAPHY

Ayesha Khanbu is a final year engineering student pursuing a bachelors degree in The Department of Computer Engineering from Fr.C.R.I.T., Vashi, University of Mumbai. Her areas of interest are Parallel Computing and Compiler Construction.

Nida Patankar is a final year engineering student pursuing a bachelors degree in The Department of Computer Engineering from Fr.C.R.I.T., Vashi, University of Mumbai. Her areas of interest are Databases, Operating Systems, Computer Graphics.

Harshala Bhoir is a final year engineering student pursuing a bachelors degree in The Department of Computer Engineering from Fr.C.R.I.T., Vashi, University of Mumbai. Her areas of interest are Databases.

Amroz Siddiqui received his MTech degree in Com-puter Engineering from VJTI, Mumbai, in 2015. He is currently working as an Assistant Professor in Department of Computer Engineering in Fr. C. Ro-drigues Institute of Technology, Vashi, Navi Mum-bai. His research interests include writing parallel versions of various algorithms, optimizing compilers and automatic parallelization of C code, program-ming languages, data mining, system and network security.

